Get Func<>-y: Delegates in .NET

An overview of delegates, Func<T>, and Action<T> by JeremyBytes.com

Overview

Delegates in .NET allow us to work with methods in interesting ways. We can create callbacks, implement event handlers, and pass functions as method parameters. We can define our own delegates, use delegates that are defined in the framework libraries, or even use the shortcuts `Func<T>` and `Action<T>`.

First, we'll take a look at what delegates are. Then we'll use an example to see how we can use delegates in our application code. Finally, we'll see how `Func<T>` and `Action<T>` can simplify that code.

What is a Delegate?

From the Visual Studio documentation: "A delegate is a type that defines a method signature." Many types in .NET define data of some sort (such as `DateTime` or `int`). Most types are classes that contain both data and method implementations (such as a `Button` or `List<T>`). A delegate is a bit different in that it defines a signature for a method. Then a delegate variable can be created with this delegate type. At that point, we can assign any method that matches the defined signature to the delegate variable. Sounds pretty confusing, doesn't it?

Let's take a look at some delegates in action. This will help us get a handle on how they are defined, how to use them, and why we would want to use them. We'll see a couple of features along the way such as decoupling of callers from targets, passing methods as parameters, and taking advantage of multicast capabilities.

The Set Up

We'll start with a skeleton of a WPF project. You can download the source code for the application here: http://www.jeremybytes.com/Downloads.aspx. The sample code we'll be looking at is built using .NET 4 and Visual Studio 2010 (however, everything will work with .NET 3.5 and Visual Studio 2008). The download includes the starter application and the completed code. The starter application contains the following.

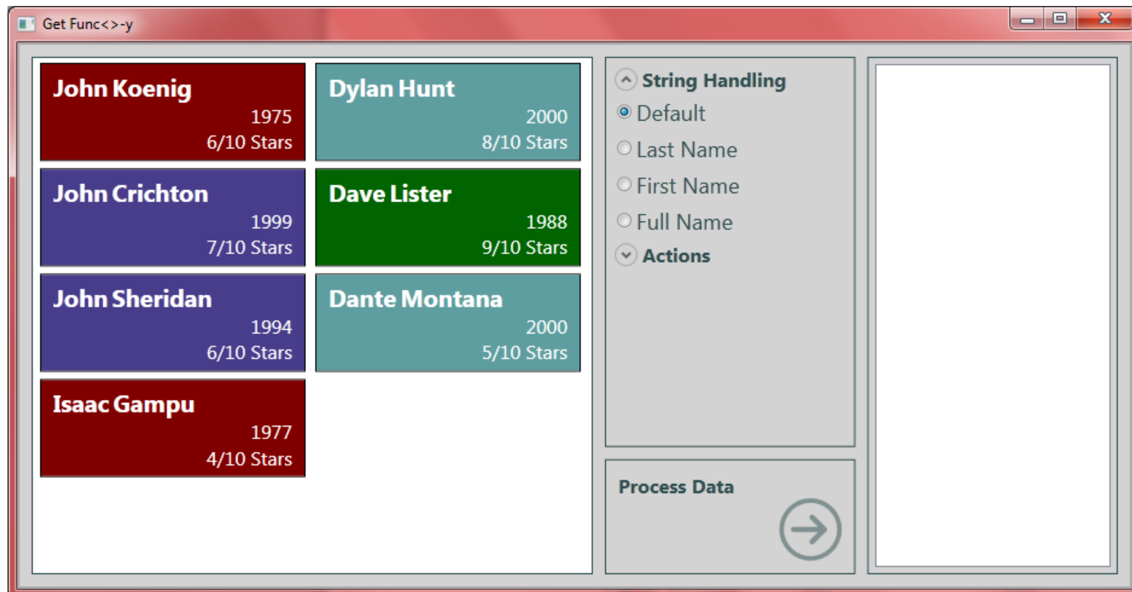
The Project

The solution contains a single project: `FuncActionDelegates`. This was created by creating a new WPF application.

©Jeremy Clark 2011

Updates

In the WPF application, the user interface has been implemented in the `MainWindow.xaml` window. Here's the running application:



The code-behind page (`MainPage.xaml.cs`) has a line of code in the constructor to get a list of people objects to load into the list box on the left (see `Person` class below for more info). In addition, the button click event for the "Process Data" button has been stubbed out with a single line to clear the output list box on the lower right. Here's the code:

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
        PersonListBox.ItemsSource = People.GetPeople();
    }

    private void ProcessDataButton_Click(object sender, RoutedEventArgs e)
    {
        OutputList.Items.Clear();
    }
}
```

The project also contains a `Converters.cs` file that contains a number of value converters that are used in the display. If you are interested in the XAML and value converters in this project, you can look up the *Introduction to Data Templates and Value Converters in Silverlight* demo and sample code on the website: http://www.jeremybytes.com/Downloads.aspx (note: this works perfectly well in WPF as well).

The `Person.cs` file contains 2 simple classes: a static `People` class and a `Person` class:

```
public static class People
{
    public static List<Person> GetPeople()
    {
        var p = new List<Person>()
        {
            new Person() { FirstName="John", LastName="Koenig",
                StartDate = DateTime.Parse("10/17/1975"), Rating=6 },
            ... other hard-coded records removed for brevity
        };
        return p;
    }
}

public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime StartDate { get; set; }
    public int Rating { get; set; }

    public override string ToString()
    {
        return string.Format("{0} {1}", FirstName, LastName);
    }
}
```

As we can see here, the `People` class is a static class with a single method called `GetPeople`. This returns a hard-coded list of `Person` objects. The `Person` class consists of 4 properties and an override of the `ToString()` method. We'll be adding to the `Person` class in just a little bit.

So with that in place, let's get started!

Implementing String Handling

In the UI, we have a set of radio buttons under "String Handling". When we click the "Process Data" button, we want to output the people from the list to the output list on the right. But here's the catch: we want to vary the output based on which radio button is selected. We will do this by defining and using a delegate.

From the definition we have above, we see that a delegate is simply a type that defines a method signature. We create this like any other type by specifying an access modifier (public, private, internal, etc.), using the `delegate` keyword, and then defining the signature. Let's define a `PersonFormat` delegate type that we can use for the string handling. I put this in the `Person.cs` file so we can see it easily, but this can be defined anywhere in the namespace:

```
namespace FuncActionDelegates
{
    public static class People...

    public delegate string PersonFormat(Person input);

    public class Person...
}
```

This method signature defines a single parameter of type `Person` and returns a `string`. Note that the delegate is declared outside of the `Person` class so that it will be available anywhere in our namespace.

Next, we use the delegate in the `Person` class to create a custom `ToString()` method:

```
    public class Person
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public DateTime StartDate { get; set; }
        public int Rating { get; set; }

        public override string ToString()
        {
            return string.Format("{0} {1}", FirstName, LastName);
        }

        public string ToString(PersonFormat format)
        {
            return format(this);
        }
    }
```

What this allows us to do is pass a method as a parameter to another method. Our parameter is of type `PersonFormat` (our delegate) which means that we can pass in a method that matches the signature we defined above. In the `ToString` method body, we are invoking the delegate by calling the delegate and passing the parameter. In our case, we call `format` (the name of our parameter) and pass in `this` (which is the instance of the `Person` class). Once we pull a few more pieces together, this should become clear.

Now, we'll switch over to the code-behind for the XAML. Inside the `Window` class, we'll create a method that matches the signature of our delegate.

```
  public partial class MainWindow : Window
  {

      public string OutputLastName(Person input)
      {
          return input.LastName;
      }
```

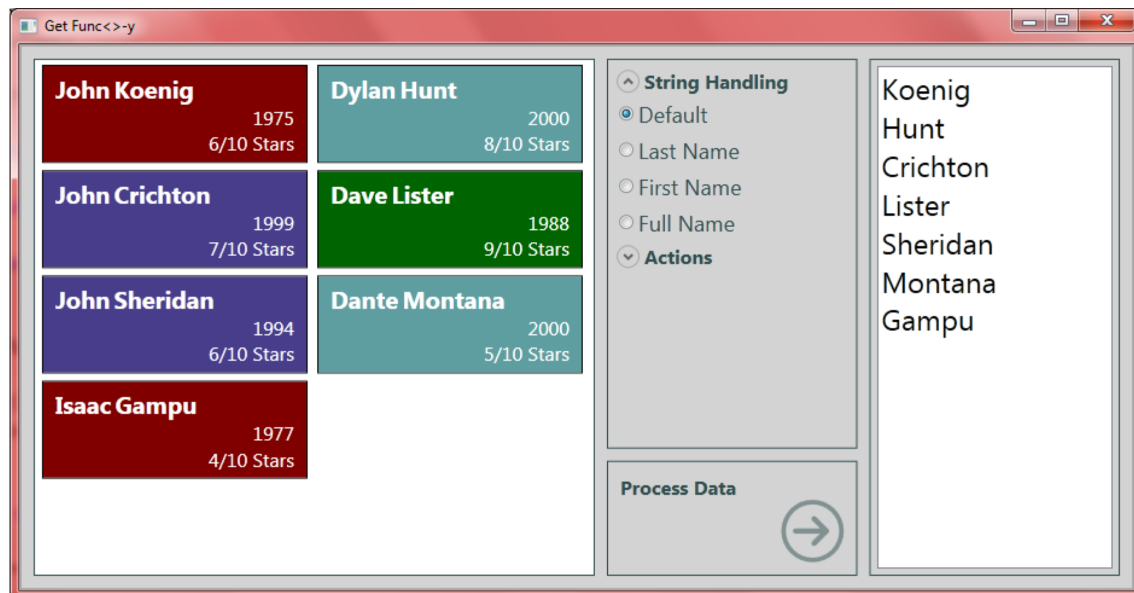

©Jeremy Clark 2011

```
public MainWindow()
{
    InitializeComponent();
    PersonListBox.ItemsSource = People.GetPeople();
}

private void ProcessDataButton_Click(object sender, RoutedEventArgs e)
{
    OutputList.Items.Clear();
    foreach (Person per in PersonListBox.Items)
        OutputList.Items.Add(per.ToString(OutputLastName));
}
}
```

First, we create a method that conforms to the delegate signature: `OutputLastName`. Note that `OutputLastName` takes a single parameter of type `Person` and returns a `string` – exactly the same signature as our delegate. In this simple method, we will output the `LastName` property of the `Person`.

Down in the button click event handler, we have a `foreach` loop to iterate through each of our `Person` objects and then add them to the output list. Note that for the `per.ToString()` method, we are calling the version that takes a delegate parameter. For the parameter, we are passing the `OutputLastName` method. This is just the method name (without parentheses). This is how you pass a method as a parameter.

Now if we run the application and click the button, we get the following output:



Seems like quite a lot of work just to get this output. But soon we'll start taking advantage of the power of delegates.

In addition to the delegate type (`PersonFormat`), we can also create delegate variables. These are instances of the type and are declared just like any other variable. Here, we'll create a delegate variable called `proc` and use it in our `ToString` method.

```
public partial class MainWindow : Window
{
    PersonFormat proc;

    public string OutputLastName(Person input)
    {
        return input.LastName;
    }

    public MainWindow()
    {
        InitializeComponent();
        PersonListBox.ItemsSource = People.GetPeople();
    }

    private void ProcessDataButton_Click(object sender, RoutedEventArgs e)
    {
        OutputList.Items.Clear();
        proc = OutputLastName;
        foreach (Person per in PersonListBox.Items)
            OutputList.Items.Add(per.ToString(proc));
    }
}
```

We can see that we declare the `proc` variable by specifying its type just like any other variable. Then in the Click event, we assign our method to the variable and then use the variable as the `ToString` parameter. If we run the application, we will see the same output that we had before.

Next steps will be to harness the power of delegates by moving the decision of which method to pass from compile-time to run-time.

> *Note: we are putting quite a bit of our code in the code-behind for the XAML. This is generally not the preferred place for this code. In most applications we would have this broken out into our business layer or UI controller/view model. But for our purpose of showing how delegates work, we'll take the all-in-one approach for simplicity.*

Choosing a Method to Assign

Let's implement the radio buttons so that we can select how we want our output to display. To make things a little easier, we'll move the formatting method (`OutputLastName`) to a separate static class (we'll give it a slightly different name at the same time). The static class will allow us to easily add more methods without cluttering up our code.

We'll create a new class called `Formatters` that will look like the following:

```
namespace FuncActionDelegates
{
    public static class Formatters
    {
        public static string Default(Person input)
        {
          return input.ToString();
        }

        public static string LastNameToUpper(Person input)
        {
          return input.LastName.ToUpper();
        }

        public static string FirstNameToLower(Person input)
        {
          return input.FirstName.ToLower();
        }

        public static string FullName(Person input)
        {
          return string.Format("{0}, {1}", input.LastName, input.FirstName);
        }
    }
}
```

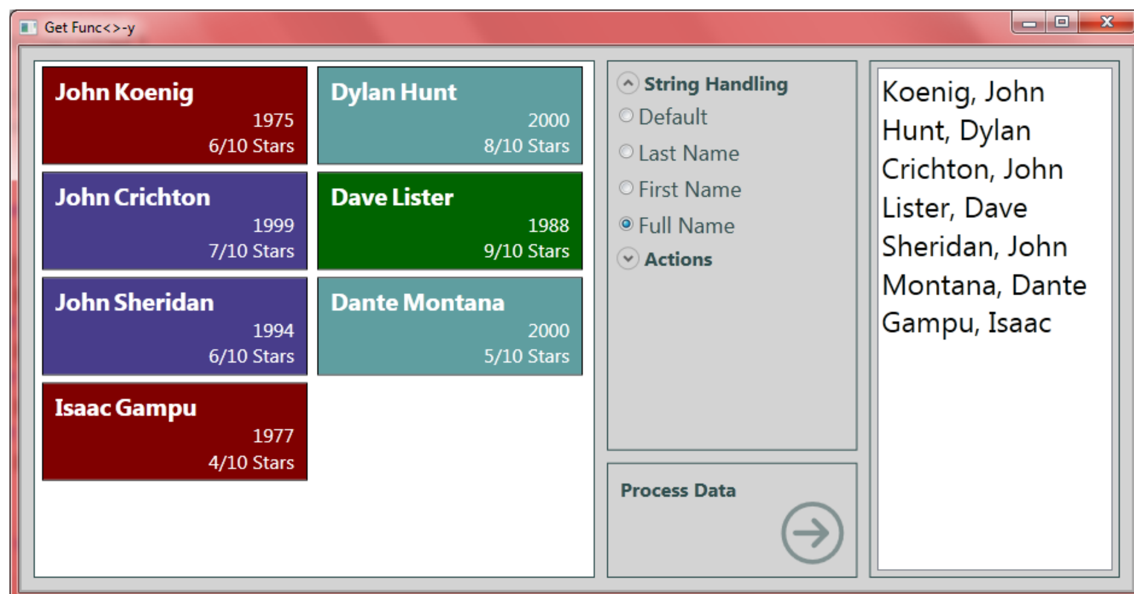
We have 4 different methods here (that match our radio buttons in the UI). Notice that they all conform to the same method signature – they all take a `Person` as parameter and return `string`. Now all we need to do is select which of these methods to assign at runtime. Let's go back to the XAML code-behind.

```
  public partial class MainWindow : Window
  {
      PersonFormat proc;

      public MainWindow()
      {
          InitializeComponent();
          PersonListBox.ItemsSource = People.GetPeople();
      }

      private void AssignDelegate()
      {
          if (Option1Button.IsChecked.Value)
              proc = Formatters.Default;
          else if (Option2Button.IsChecked.Value)
              proc = Formatters.LastNameToUpper;
          else if (Option3Button.IsChecked.Value)
              proc = Formatters.FirstNameToLower;
          else if (Option4Button.IsChecked.Value)
              proc = Formatters.FullName;
      }
```

```
        private void ProcessDataButton_Click(object sender, RoutedEventArgs e)
        {
            OutputList.Items.Clear();
            AssignDelegate();
            foreach (Person per in PersonListBox.Items)
                OutputList.Items.Add(per.ToString(proc));
        }
    }
```

A few updates here. First, we removed the `OutputLastName` method that we had previously defined. Next, we added an `AssignDelegate` method. Its responsibility is to check to see which radio button is checked and then assign the appropriate method to our delegate variable. You can see how our static class allows us to easily reference the methods and assign the appropriate one.

Finally, in the button click event, we changed the assignment to the `proc` variable to a call to the `AssignDelegate` method. Now when we run the application, the output is determined by which radio button is selected. Here's a sample with the "Full Name" option selected:



Decoupling

So what we have now is an application that uses a delegate to pass a method as a parameter. We have decoupled the formatting of an output string from the `Person` class itself. The `Person` class is no longer concerned with handling the output formats; that has been externalized by relying on the delegate to handle the actual formatting.

Wherever we use the `Person` class, we can now pass in any method we want to control the output of the `ToString(PersonFormat format)` method, and the `Person` class does not need to be directly aware of the output formatting. (Technically, we cannot pass in any method we want, but we can use

any method that conforms to the delegate signature of taking a `Person` as a parameter and returning a `string`.) Additionally, if we want to add new output formats, we do not need to modify the `Person` class.

We could even distribute the Person class in a compiled assembly, and the developer using that assembly could still extend the output formatting because of the delegate.

A Bit of Error Handling

Before going any further, we need to take a look at some error handling. In the `Person.ToString(PersonFormat format)` method body, we are invoking the delegate. But we have a problem: what happens if we pass `null` to the method?

If we try this, then we get a `NullReferenceException`. This means that we need to check for null values before invoking the delegate. Here is the updated code:

```
        public string ToString(PersonFormat format)
        {
            if (format != null)
                return format(this);
            return string.Empty;
        }
```

Now, if a null is passed in, the method simply returns an empty string. Otherwise, it will run the delegate using `this` (the Person instance) as the parameter.

It's Time to Get Func<>-y

Now that we have seen a few basics of defining delegates, let's take a look at how `Func<>` can help us make our code a bit more concise by using a predefined delegate from the .NET framework.

If we look at the Visual Studio Help for `Func<T, TResult>`, we have the following definition: "Encapsulates a method that has one parameter and returns a value of the type specified by the TResult parameter."   And the syntax is defined as follows:

```
public delegate TResult Func<in T, out TResult>(
    T arg
)
```

What this means is that `Func<T, TResult>` is a delegate that expects a `T` as the parameter, and `TResult` as the return type. Let's compare this side by side with our delegate:

```
    public delegate string  PersonFormat            (Person input);
    public delegate TResult Func<in T, out TResult> (T arg);
```

The declaration of `Func<>` is exactly like the declaration of our `PersonFormat` delegate. The exception is that `Func<>` adds generic types for the parameter and return value. So, `TResult` is our return type,

and `T` is our parameter type. The good news is that `Func<>` is built in to the .NET framework, so we can take advantage of it without any additional declarations.

There are also other versions of `Func<>`, such as `Func<T1, T2, TResult>` which takes 2 parameters and returns a result. In .NET 3.5, this goes up to a total of 4 parameters (T1 – T4); .NET 4.0 has additional methods that go all the way up to 16 parameters (T1 – T16) – although if you have a method that needs 16 parameters, you may want to rethink your code a bit. The important thing to remember is that the last item in the generic list is the return type; everything else is parameters.

In our example, our delegate takes a `Person` parameter and returns a `string`. With `Func<T, TResult>`, we do not need to explicitly define the `PersonFormat` delegate type. Instead, we can simply use `Func<Person, string>`. There are 3 places to update our code.

1. Remove the declaration of `PersonFormat` (we'll just comment it out) in our `Person.cs` file:

   ```
   //public delegate string PersonFormat(Person input);
   ```

2. Update the parameter type of the `ToString()` method in the `Person` class from `PersonFormat` to a `Func<>` type that takes a `Person` parameter and returns a `string`:

   ```
   public string ToString(Func<Person, string> format)
   {
       if (format != null)
           return format(this);
       return string.Empty;
   }
   ```

3. Change the type of the `proc` variable in our `MainWindow.xaml.cs` file from `PersonFormat` to the appropriate `Func<>` type:

   ```
   //PersonFormat proc;
   Func<Person, string> proc;
   ```

If we run the application again, we see that we get exactly the same results. Changing the delegate from our custom type to `Func<>` did not change the functionality. The signatures still match, so we can use the same formatting methods as before.

Why Func<>?

So, the big question is why would we want to do this? What advantages does using `Func<>` have over the explicitly defined delegate we started with?

First, it tightens up our code; we don't need to explicitly define a delegate. It also makes our code easier to read. Everywhere we use the delegate, we use `Func<Person, string>` which gives us all of the information we need (including parameters and return types). When we use the explicit delegate (`PersonFormat`), we need to look up the definition of the type in order to find out the parameter types and return type.

Next, `Func<>` lends itself to using lambda expressions. We'll look at this in just a little bit.

Most importantly, `Func<>` resolves interoperability issues with delegate variables. When we create an explicit delegate, it is a new type. This type is not directly interchangeable with other delegates even if they have the same method signature. This will make a little more sense with a quick example.

First, let's declare 2 delegate types with the same signature:

```
public delegate string PersonFormat(Person input);
public delegate string OtherPersonFormat (Person input);
```

Then, we'll declare 2 delegate variables:

```
PersonFormat delegate1;
OtherPersonFormat delegate2;
```

Then we'll try to assign one delegate variable to the other:

```
delegate1 = delegate2;
```

This doesn't work. Even though the delegates have the same signature (a `Person` parameter and `string` return type), they are completely separate and incompatible. If we try to build the application, we get the following error: "Cannot implicitly convert type 'FuncActionDelegates.OtherPersonFormat' to 'FuncActionDelegates.PersonFormat'".

There is a way around this by using the `new` operator and a cast, but it doesn't usually make sense to do it this way. Instead, we use `Func<Person, string>`, and the 2 variables are now of the same type. As such, they are compatible and can be assigned to each other and used interchangeably.

Updating to Anonymous Delegates

Back to our sample application, our `Formatters` static class contains a set of named delegates. We can easily change these to anonymous delegates. To convert to anonymous delegates we in-line the code where we do the assignment. We use the `delegate` keyword, then add the parameters and method body. Here's the update to our `AssignDelegate()` method:

```
private void AssignDelegate()
{
  if (Option1Button.IsChecked.Value)
      proc = delegate(Person input)
      {
        return input.ToString();
      };
  else if (Option2Button.IsChecked.Value)
      proc = delegate(Person input)
      {
        return input.LastName.ToUpper();
      };
```

```
    else if (Option3Button.IsChecked.Value)
        proc = delegate(Person input)
        {
          return input.FirstName.ToLower();
        };
    else if (Option4Button.IsChecked.Value)
        proc = delegate(Person input)
        {
           return string.Format("{0}, {1}", input.LastName, input.FirstName);
        };
}
```

Notice that these take the methods that were in the separate class and include them all in this method, using the `delegate` keyword and then the parameters and the method body. This is useful for one-time-use delegates. As long as this is the only place we want to use these particular delegates, then this often makes the code more readable and easier to follow (especially when we use lambda expressions, which we will do in just a bit).

We are no longer using the `Formatters` class that we created earlier. We can comment out this entire class (or event delete the file from the project), and our application still works exactly as it did before.

Lambda Expressions and Func<>

Lambda expressions and `Func<>` are made for each other (literally – they were both added to the framework at the same time to support LINQ). The rule of thumb that I use is that anywhere I see `Func<>`, it means "put your lambda expression here." Lambda expressions themselves are a larger topic; for a bit more information, you can look up the *Learn to Love Lambdas* demo and sample code on the website: http://www.jeremybytes.com/Downloads.aspx.

The short version is that lambda expressions are anonymous delegates, so we can update our syntax from our anonymous delegates to lambda expressions. We'll take a couple of steps to convert the anonymous delegates to lambda expressions, and then use some of the syntactical goodies that lambda expressions offer us.

We'll use the first `if` statement from our `AssignDelegate()` method for our conversion. Here's the original:

```
      if (Option1Button.IsChecked.Value)
        proc = delegate(Person input)
        {
          return input.ToString();
        };
```

To convert this to a lambda expression, we just need to remove the `delegate` keyword and add the "=>" operator (the "goes to" operator):

```
if (Option1Button.IsChecked.Value)
    proc = (Person input) =>
    {
        return input.ToString();
    };
```

Next, for lambda expressions, the parameter type is optional:

```
if (Option1Button.IsChecked.Value)
    proc = (input) => { return input.ToString(); };
```

If there is a single parameter, then the parentheses are optional. In addition, since we have an expression lambda, we can remove the `return` keyword. And since the body consists of a single statement, we can remove the curly braces. Here's what we end up with:

```
if (Option1Button.IsChecked.Value)
    proc = input => input.ToString();
```

Finally, it's common to use single character variable names with lambda expressions. We'll use "`p`" to stand for the `Person` type of our parameter (but the name really doesn't matter):

```
if (Option1Button.IsChecked.Value)
    proc = p => p.ToString();
```

After we do the same with the rest of our delegate assignments, here's what our `AssignDelegate()` method looks like:

```
private void AssignDelegate()
{
    if (Option1Button.IsChecked.Value)
        proc = p => p.ToString();
    else if (Option2Button.IsChecked.Value)
        proc = p => p.LastName.ToUpper();
    else if (Option3Button.IsChecked.Value)
        proc = p =>  p.FirstName.ToLower();
    else if (Option4Button.IsChecked.Value)
        proc = p => string.Format("{0}, {1}", p.LastName, p.FirstName);
}
```

Notice how much more compact this is compared to the previous version of the `AssignDelegate()` method. And assuming that we are comfortable with lambda expressions, this is very easy to read as well. (Yes, I realize that is a big "if" – but lambda expressions are a great thing to become comfortable with.)

Time to Get Action<>-y

Okay, so "Get Action<>-y" doesn't have the same ring to it as "Get Func<>-y". But `Action<T>` is just as useful in the realm of delegates as `Func<T>`. If we look at the Visual Studio help for `Action<T>`, we have the following definition: "Encapsulates a method that has a single parameter and does not return a value." And the syntax is defined as follows:

```
public delegate void Action<in T>(
    T obj
)
```

So, basically, this works almost exactly like `Func<>`, with the difference being that `Action<>` does not return a value (return type is `void`). It is designed to perform some sort of action rather than return a value. And just like `Func<>`, there are additional versions of `Action<>` that take multiple parameters: up to 4 parameters in .NET 3.5, and up to 16 parameters in .NET 4.0.

As a side note, both `Action<>` and `Func<>` have versions that take zero parameters. These are useful in some situations, but you will generally use a version that takes at least 1 parameter.

Multicast Delegates

All delegates in .NET are multicast delegates. This means that we can assign more than one method to a single delegate variable. When the delegate is invoked, all of the assigned methods in the method list are run. To add a method to a delegate variable, we use the "`+=`" operator. We are familiar with this syntax when assigning event handlers (and event handlers are just a special type of delegate).

Assigning multiple methods to a single delegate variable makes more sense when using a delegate with `void` return type. Think about this: if we have a delegate that returns a string, and multiple methods are assigned (which all return string), then which string is actually returned? It turns out that the string that is returned is the return value of the last method run (which usually isn't very useful). Because of this, we usually have single-assignment for delegates when we want to use the return value.

But for delegates that perform an action and do not return a value, it is often useful to take advantage of the multicast capabilities of delegates. We'll be looking at a fairly trivial example here, but it will give us the generally idea of how it works.

A Little Housekeeping

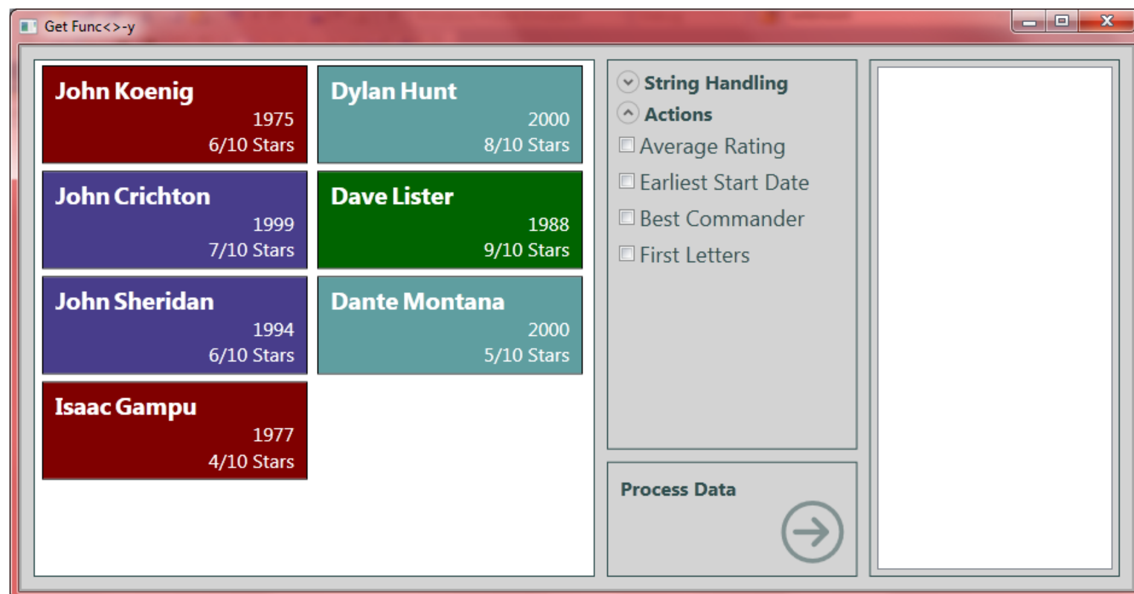
First, let's do a little bit of housekeeping that will control how our UI works. Right now, whenever the "Process Data" button is clicked, the "String Handling" section is run. We will be adding code to take care of the "Actions" section. Both of these sections are expander controls. Let's add some code that only runs the code from the section if the expander is actually expanded.

Here's the updated button click handler:

```
private void ProcessDataButton_Click(object sender, RoutedEventArgs e)
{
    OutputList.Items.Clear();

    if (StringExpander.IsExpanded)
    {
        AssignDelegate();
        foreach (Person per in PersonListBox.Items)
            OutputList.Items.Add(per.ToString(proc));
    }
    if (ActionExpander.IsExpanded)
    {
        // do stuff here
    }
}
```

Adding the Actions

Let's take a look at the Actions we have available from our UI:



Notice that the options on the Action panel are checkboxes. This means that we can have multiple options checked at the same time. Here's a quick overview of what each checkbox will do:

- **Average Rating**: Calculate the average of the "Rating" properties **from the list** and display it in the output list.
- **Earliest Start Date**: Get the "Start Date" property **from the list** with the earliest value and display it in the output list.
- **Best Commander**: Get the "Last Name" property for the item with the highest "Rating" **from the list** and display it in a message box.
- **First Letters**: Get the first letter of the "Last Name" property for each item **from the list** and display it in the console window.

Each of these functions relies on a List<Person> parameter. What is done with the calculated values is determined separately by each function. If we were to explicitly define a delegate type, it would look like this:

```
public delegate void PeopleAction(List<Person> input);
```

But, since we want to use Action<T> and save us the trouble of the explicit delegate type, our delegate variable will look like this instead:

```
Action<List<Person>> act;
```

If we were to walk up to this definition without any context, it can be confusing, especially since we have nested generic types. But since we are comfortable with how Func<> and Action<> are used, we know that the method signature for this would take a single parameter (of type List<Person>) and return void.

Assigning the Actions

Now, we'll create an AssignAction() method similar to the AssignDelegate() method that we have. We'll skip the named delegates and anonymous delegates and go straight to the lambda expressions. We'll start out with just the shell to get a feel for the Action<> delegate.

```
private void AssignAction()
{
    act += p => OutputList.Items.Add(p.Average(r => r.Rating));
}
```

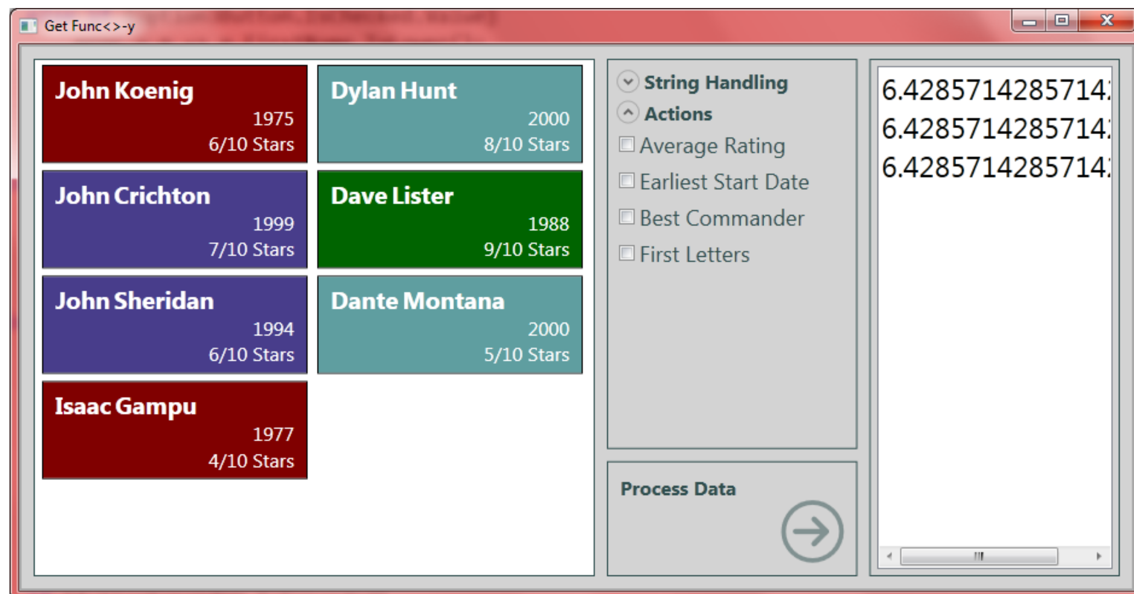
This delegate is not returning a value; instead, it is performing an action. In this case, it is adding an item to the output list. That item is the average of the Rating fields for each Person in the list. One other thing to notice is that we are using the "+=" operator and not simply the equals. This is how we take advantage of the multicast capability, and we'll see the effect of that in just a moment.

We'll use the delegate in the placeholder section of our button Click event:

```
private void ProcessDataButton_Click(object sender, RoutedEventArgs e)
{
    OutputList.Items.Clear();

    if (StringExpander.IsExpanded)
    {
        AssignDelegate();
        foreach (Person per in PersonListBox.Items)
            OutputList.Items.Add(per.ToString(proc));
    }
    if (ActionExpander.IsExpanded)
    {
        AssignAction();
        act(People.GetPeople());
    }
}
```

As we did for the string handling section, we call AssignAction() to assign a value to our act delegate variable. Then we invoke the delegate by calling act with a parameter. In this case, we call the static method People.GetPeople() to get a List<Person>.

If we run the application now, expand the "Actions" panel, and then click the "Process Data" button. We will see the average rating in the output list. If we click the "Process Data" button three times, we get the following output:



At first glance, this looks like we forgot to clear the output box. However, if we double-check the button Click code, we see that the first statement is OutputList.Items.Clear(). So what's going on?

The first time we click the button, the AssignAction() method assigns our lambda expression to the delegate variable. The "+=" operator actually appends to what is already there. The second time we click the button, our lambda expression is assigned **again** to the delegate variable. So there are now two methods assigned. The third time, we have three methods assigned to our variable. This is showing the multicasting in action. We are still invoking the delegate a single time (in the button click handler), but the variable is running all three methods that are assigned.

To fix this behavior, we can just set the variable to null at the top of the AssignAction() method:

```
private void AssignAction()
{
    act = null;
    act += p => OutputList.Items.Add(p.Average(r => r.Rating));
}
```

Now if we run the application again, no matter how many times we click the button, we still only get one item in our output list.

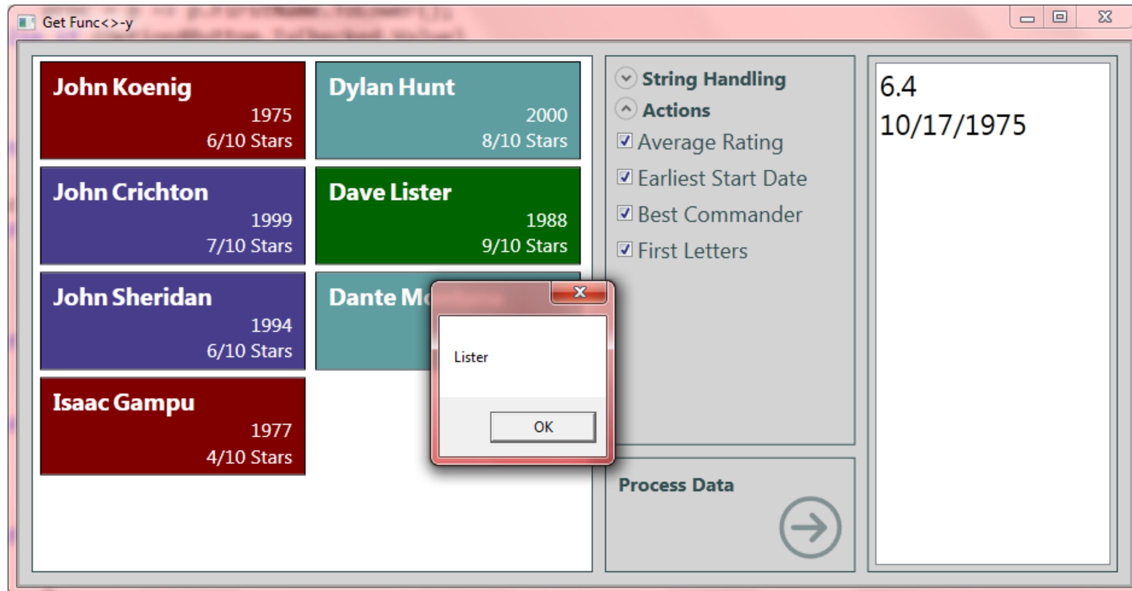
©Jeremy Clark 2011

Now we'll update the `AssignAction()` method so that it examines the values of the checkboxes before assigning the delegate, and we'll add in our other methods (and add a formatting string to the first method).

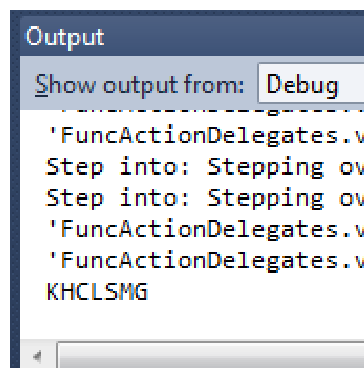
```
private void AssignAction()
{
    act = null;
    if (OptionAButton.IsChecked.Value)
        act += p => OutputList.Items.Add(
            p.Average(r => r.Rating).ToString("#.#"));


    if (OptionBButton.IsChecked.Value)
        act += p => OutputList.Items.Add(p.Min(s => s.StartDate));

    if (OptionCButton.IsChecked.Value)
        act += p => MessageBox.Show(
            p.OrderByDescending(r => r.Rating).First().LastName);

    if (OptionDButton.IsChecked.Value)
        act += p =>
            {
                p.ForEach(c => Console.Write(c.LastName[0]));
                Console.WriteLine();
            };
}
```

Each of the methods that are assigned here does not return a value, but it does perform an action. The first two calculate a value and add it to the output list. The third calculates a value and displays it in a message box. The last calculates a value and then outputs it to the console window (this shows up in the Ouput window in Visual Studio when you run in debug mode). I won't go into detail on how the items are calculated; this is all part of the LINQ goodness in the .NET Framework and is a good place to explore further.

Now, when we run the application, the output will depend upon which items are checked. Here's an example of the output from the application:



And the Output window in Visual Studio:

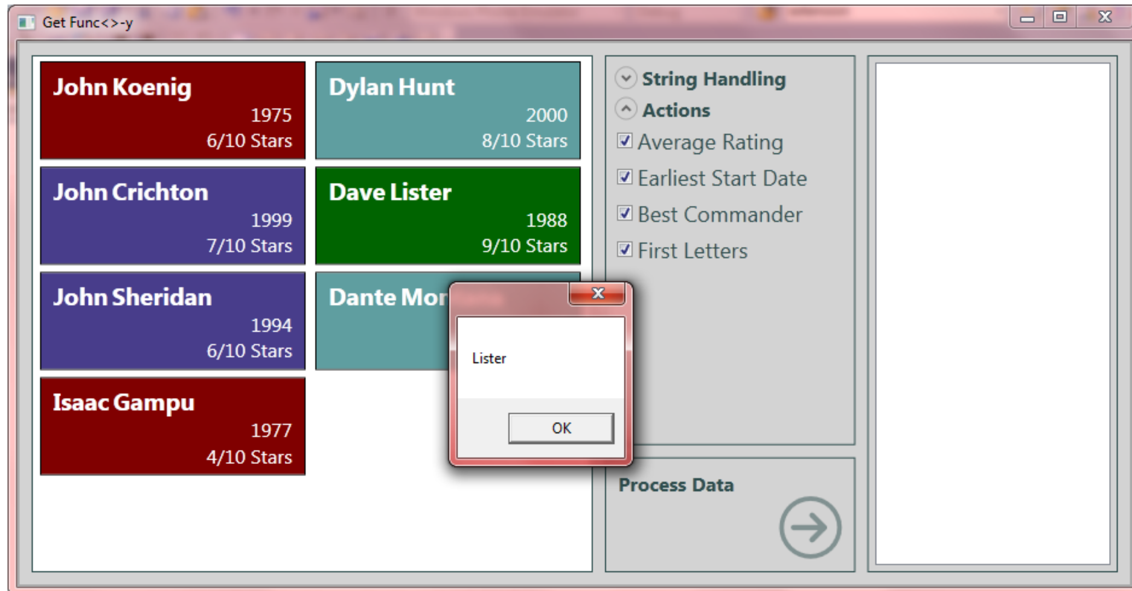


Since we had all 4 checkboxes checked, we see the average rating in the output list (6.4), the earliest start date in the output list (10/17/1975), the highest-rated commander in a message box (Lister), and the first letters of the last names in the Output window (KHCLSMG).

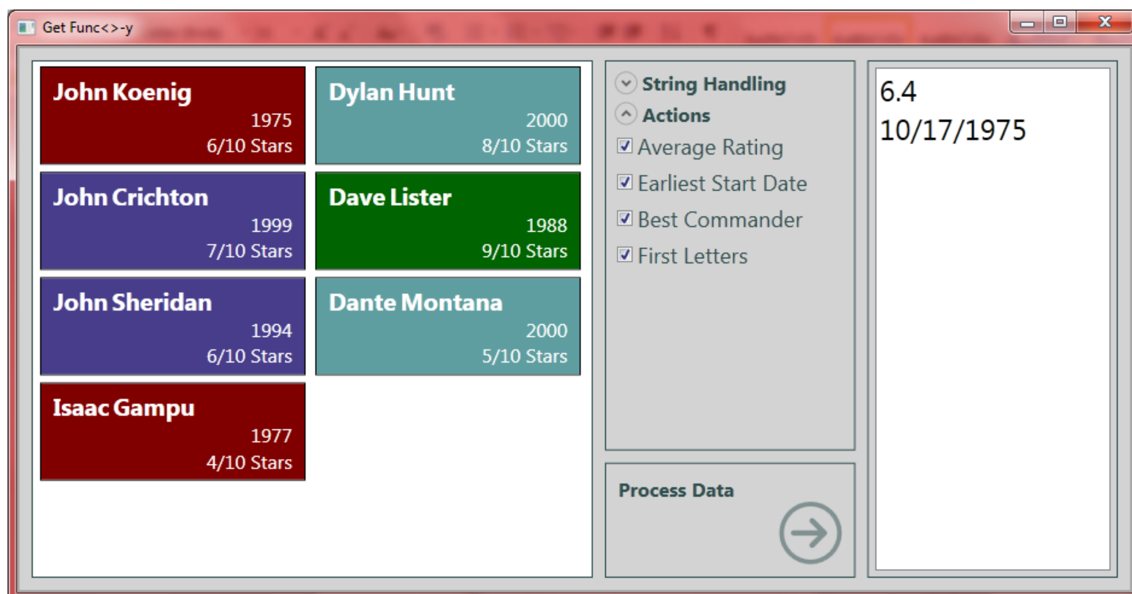
All 4 of these were run with a single invocation of our `act` delegate variable in the button click event handler.

The delegates are invoked in the order that they are assigned. There is no magic multi-threading going on here. After a method is run to completion, then next method in the list is run. We can see the behavior more clearly if we change the order. Move the "`OptionCButton`" statement (the one with the `MessageBox.Show` call) to the top of the list. Since `MessageBox.Show()` is a modal function, the application will stop processing additional requests until after the dialog is cleared.

Now if we were to check all 4 boxes, we get the following output (before clicking the OK button in the dialog):



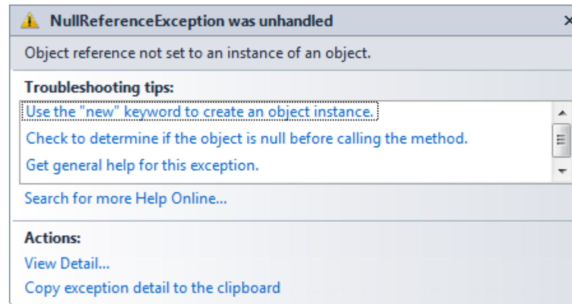
And after clicking "OK":



The output for the Average Rating and Earliest Start Date do not show up in the output list until **after** the dialog is cleared. This is because the `MessageBox.Show()` method is called first; then the other two methods are called – but only after the first method has completed. We would see the same type of behavior by looking for the output to show up in the Console window.

More Error Handling

There is still one problem that we have: what if we do not check any checkboxes when we run the application?



As we saw before, we get a NullReferenceException. To protect against this, we just need to check if our delegate variable is populated before we invoke it:

```
if (ActionExpander.IsExpanded)
{
    AssignAction();
    if (act != null)
        act(People.GetPeople());
}
```

Now, if there are no items checked, the delegate does not get invoked.

How Func<>-y Can We Get?

We've seen a number of features for delegates, `Func<>`, and `Action<>`. There's also a huge world to explore beyond this introduction.

Decoupling Code

Decoupling the calling method from the target method can add a layer of abstraction to our code. In our case, the button click event handler invokes the delegate. The button click event itself does not make decisions and call methods directly. It relies on other methods to populate the delegate and then works simply with that. We can easily imagine being able to extend this by assigning methods to delegates in other assemblies or by using a form of dependency injection (DI).

Methods as Parameters

Passing a method as a parameter helps us keep our classes simple and more easily adhere to the single responsibility principle (for more on the single responsibility principle, you can do a search for Robert C. Martin and the SOLID principles). In our case, the `Person` class does not know anything about formatting output for the `ToString(Func<Person, string>)` method. The formatting has been completely externalized from the class.

I have also seen people use `Func<>` in parameters as a form of inversion of control (IoC). In that particular case, a save method took a `Func<T>` that returned a repository class. The save method could pass different methods at runtime depending on the repository that needed to be used. (This also assisted with adding seams for unit testing.)

Multicast Capabilities

As we saw with our `Action<>` delegate, it is easy to take advantage of the multicast capabilities of delegates. This allows us to run multiple methods with a single call (when the delegate is invoked). This can also be useful for a broadcast message or an implementation of the Observer pattern, where a class sends out notifications to subscribers when something changes.

Callbacks and Event Handlers

Delegates are also great for creating callbacks and event handlers. Although we did not have time to cover this topic, callbacks and event handlers are big uses for delegates. Callbacks allow us to pass a notification method into another method, letting us inject code that will run after the method has completed. This is often used in an asynchronous environment where the caller needs to be notified when a method has completed; in addition it provides a way to pass return values back from an asynchronous method.

Lambda Expressions

Lambda expressions are simply anonymous delegates. So understanding delegates helps us better understand where we can use lambda expressions. And again, in many places where you see `Func<>` being used as a parameter, you can treat it as a big sign that says, "Put your lambda expression here."

LINQ

Another topic we did not cover here is LINQ. LINQ is an extremely useful technology. If we look at the extension methods on `IEnumerable<T>`, we see dozens and dozens of methods that are available to us. The majority of these methods take some type of `Func<>` as a parameter. With a good understanding of delegates and `Func<>`, we are better equipped to understand the method signatures.

Wrap Up

So, delegates have a variety of uses. And `Func<>` and `Action<>` make using delegates in our code both quick to implement and easy to read. We've only looked at a couple of uses of delegates so that we can get an idea of how they work, but there are many others. With a basic understanding, we are better prepared when we run across these constructs in code samples or third-party libraries that we want to work with. All in all, these are a great addition to the developer toolbox.

Happy coding!

©Jeremy Clark 2011